QBUS6850: Tutorial 13 – Recommendation Systems

Objectives

- To learn how to use Surprise library
- To get familiar with recommendation systems

1. Installing Surprise

Surprise is a library for evaluating Recommendation algorithms. It can guide you in selecting the appropriate Recommendation algorithm for your particular task.

It provides:

- inbuilt sample datasets
- baseline, nearest neighbour and factorisation algorithms
- parameter optimisation tools (grid search)
- evaluation tools

Homepage: http://surpriselib.com

Documentation: http://surprise.readthedocs.io/en/stable/

Suprise is not available on the conda repository listing or conda-forge.
Therefore we have to manually install it.

1. Download Suprise v1.0.3
   https://github.com/NicolasHug/Surprise/archive/v1.0.3.zip
2. Unzip the folder
3. Open Anaconda Prompt and navigate to the unzipped folder (use cd command to change directory) e.g.
   cd C:\Users\steve\Downloads\suprise
4. Run the following commands in order:
   python setup.py build_ext –i
   python setup.py install
5. Test Suprise by:
   Opening a Python session (just type "python" at the Anaconda Prompt or use Jupyter/Spyder)
   Importing a component of suprise e.g. "from surprise import SVD"

2. Using Surprise

In this example we will use Surprise's inbuilt MovieLens dataset. This dataset contains ratings of movies from users of the MovieLens website (http://movielens.org).

We will use the SVD algorithm to predict all missing ratings (infill) from the dataset using the observed ratings.

First let's load the data

```
from surprise import SVD
from surprise import KNNBasic
from surprise import BaselineOnly
from surprise import Dataset
from surprise import accuracy
from surprise import GridSearch

# Load the full dataset.
data = Dataset.load_builtin('ml-100k')
```

Now we will do a grid search using cross validation to find the optimal parameters for the SVD method.

To specify the number of cross validation folds to use in the evaluate() function you need to use the split() function of the dataset.

```
data.split(n_folds=3)

param_grid = {'n_epochs': [5, 10], 'lr_all': [0.002, 0.005]}

# Verbose = 2, this shows lots of useful output
# http://surprise.readthedocs.io/en/stable/evaluate.html
grid_search = GridSearch(SVD, param_grid, measures = ['RMSE'], verbose = 2)

# Perform the grid search
grid_search.evaluate(data)

# Print out the average RMSE for each fold and corresponding param pairs
rmse_result = grid_search.cv_results['RMSE']
param_pairs = grid_search.cv_results['params']

for i in range(len(rmse_result)):
    print("RMSE: {0}, Params: {1}".format(rmse_result[i], param_pairs[i]))
```

Once our grid search is complete we can get the best model parameters by using the best_estimator attribute.

```
# Pickup best model from grid search
algo = grid_search.best_estimator['RMSE']
```

Now we can retrain our model on the full training set. In Suprise build_full_trainset() is every data point. It doesn't actually give you a train/test split.

```
# Retrain on full set
trainset = data.build_full_trainset()
algo.train(trainset)
```

To predict a single item rating for a user/item pair you use the predict() function. The id of each user and item must be a string.

```
pred = algo.predict('374', '500')

print("Prediction Object:")
pred
```

Using a different prediction algorithm, e.g. the knn algorithm. Note that there is no optimization of k selection here.

```
# a desctiption of different similarities measurements
# http://surprise.readthedocs.io/en/stable/similarities.html

sim_options = {'name': 'cosine',
               'user_based': False  # compute  similarities between items
               }
# http://surprise.readthedocs.io/en/stable/similarities.html
#sim_options = {'name': 'pearson',
#               "user_based': True
#               }

algo_1 = KNNBasic(sim_options= sim_options)
trainset = data.build_full_trainset()
algo_1.train(trainset)

pred = algo_1.predict('374', '500')

print("Prediction Object:")
pred

print("Predicted Rating:")
pred[3]
```

Now let's use the baseline appraoch. Note that there is no optimization of parameter selection here either.

```
bsl_options = {'method': 'als',
               'n_epochs': 5,
               'reg_u': 12,
               'reg_i': 5
               }
algo_2 = BaselineOnly(bsl_options=bsl_options)

trainset = data.build_full_trainset()
algo_2.train(trainset)

pred = algo_2.predict('374', '500')

print("Prediction Object:")
pred

print("Predicted Rating:")
pred[3]
```

3. Predicting all missing entries

First lets start by visualising our matrix of all observed entries.

This matrix is quite sparse.

```
n_users = trainset.n_users
n_items = trainset.n_items

M = np.zeros((n_users, n_items))

raw_ratings = data.raw_ratings

for rating in raw_ratings:
    r = int(rating[0]) - 1
    c = int(rating[1]) - 1
    v = rating[2]

    M[r, c] = v

import matplotlib.pyplot as plt

fig = plt.figure()
plt.imshow(M)
fig
```

Now with our trained SVD algorithm we can create a list of all user/item pairs in our dataset to fill in the rating matrix.

```
# What data did we end up with?

rows, cols = np.meshgrid(np.linspace(0, n_users-1, n_users), np.linspace(0,
n_items-1, n_items), indexing = 'ij')

rows = np.ravel(rows)
cols = np.ravel(cols)

fullset = list()

for i in range(n_users * n_items):
    fullset.append( (str(int(rows[i])), str(int(cols[i])), 0) )

preds = algo.test(fullset)

X = np.zeros((n_users, n_items))

vals = [e[3] for e in preds]

X[[int(r) for r in rows], [int(c) for c in cols]] = vals

fig = plt.figure()
plt.imshow(X)
fig
```

4. Train/Test split in Surprise

To perform a train/test split in Surprise you could do something like this:

```
data = Dataset.load_builtin('ml-100k')
raw_ratings = data.raw_ratings

random.shuffle(raw_ratings)

# A = 90% of the data, B = 10% of the data
threshold = int(.9 * len(raw_ratings))
A_raw_ratings = raw_ratings[:threshold]
B_raw_ratings = raw_ratings[threshold:]

data.raw_ratings = A_raw_ratings  # data is now the set A
data.split(n_folds=3)
```

Then you can evaluate accuracy on the test set by

```
testset = data.construct_testset(B_raw_ratings)  # testset is now the set B
# use the previously selected best model
predictions = algo.test(testset)
```

Display the top 10 resutls

```
predictions[0:9]
```

5. Low-Rank Matrix Completion

Unfortunately, Surprise is missing Low-Rank Matrix Completion (LRMC) from its set of algorithms.

LRMC is incredibly powerful, despite its simplicity. It seeks to find a new matrix that is low-rank, while keeping the observed entries fixed.

Intuition

Many users of netflix have similar or shared tastes. Their individual taste can be described as a combination of many other peoples tastes. Mathematically this means that each users rating vector (their taste) is a linear combination of other users. Therefore the total rating matrix must be relatively low-rank. So we should try to find the lowest rank matrix while keeping the existing user ratings fixed.

You should notice that in the MovieLens example the final recovered matrix is low-rank (see last Figure).

Example

Below I have provided a reference implementation and an example of usage on a synthetic dataset.

First let's define a function to solve the LRMC and a helper function to solve the nuclear norm proximal problem.

```
import numpy as np
import pandas as pd
import scipy.sparse.linalg as sparse
import scipy as sp

def solve_nn(X, lam):
    # Solves the proximal nuclear norm problem
    # min lambda * |X|_* + 1/2*|X - Y|^2
    # solved by singular value shrinking

    # Cai, J. F., Candès, E. J., & Shen, Z. (2010).
    # A singular value thresholding algorithm for matrix completion.
    # SIAM Journal on Optimization, 20(4), 1956-1982.

    U, s, V = sp.linalg.svd(X, full_matrices=False)

    new_s = np.maximum(s - lam, 0)

    return U.dot(np.diag(new_s)).dot(V), new_s


def solve_lrmc(M, omega, tau = 1, mu = 0.1, rho =1 , iterations = 100, tol = 10e-6):
    # Solves the following problem
    # min_A    tau * | A |_*
    # s.t. P.*M = P.*A + P.*E

    # which we convert to the unconstrained problem
    # min_A   tau * | A |_* + < Y, P.*A - P.*M > + mu/2 | P.*A - P.*M |_F^2

    # Adapted from
    # Lin, Z., Liu, R., & Su, Z. (2011).
    # Linearized alternating direction method with adaptive penalty for low-rank
representation.
    # In Advances in neural information processing systems (pp. 612-620).

    f_vals = np.zeros(iterations)
    last_f_val = np.inf

    P = np.zeros(M.shape)
    P[omega] = 1

    Y = np.zeros(M.shape)
    A = np.zeros(M.shape)

    for k in range(iterations):
        partial = mu * (P*A - (P*M - 1/mu * Y))
        V = A - 1/rho * partial

        A, s = solve_nn(V, tau/rho)
        Y = Y + mu * (P*A - P*M)

        f_vals[k] = tau * np.sum(s)

        if (np.abs(f_vals[k] - last_f_val) <= tol):
            break
        else:
            last_f_val = f_vals[k]

    return A, f_vals
```

Functions to solve LRMC and a helped function to solve the nuclear norm proximal problem

Then create some synthetic low-rank data

```
# 200 x 100  = 200 x 5 by 5 x 100
m = 200
n = 100
r = 5

A = np.dot(np.random.rand(m, r), np.random.rand(r, n))

print(np.linalg.matrix_rank(A))

fig = plt.figure()
plt.imshow(A)
fig
```

Then randomly sample from the data. MM is our partially observed ratings matrix

```
sample_prop = 0.3

omega_linear = np.random.choice(int(m*n), int(m*n*sample_prop))

omega  = np.unravel_index(omega_linear, (m, n))

M = np.zeros(A.shape)
M[omega] = A[omega]

fig = plt.figure()
plt.imshow(M)
fig
```

Finally solve the LRMC objective

```
print("Solving...")

X, fvals = solve_lrmc(M, omega)

print("Done")
```

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, sharey=True)
ax1.imshow(A)
ax1.set_title('Original')
ax2.imshow(M)
ax2.set_title("Observed")
ax3.imshow(X)
ax3.set_title("Recovered")
fig
```

Compare the first row of A , M and X .

Note that A and X are very close.

```
print(A[0,:])

print(M[0,:])

print(X[0,:])
```